1. The functional computation model

Some history

In 1936 two computation models were introduced:

(1) Alan Turing invented a class of machines (later to be called Turing machines) and defined the notion of computable function via these machines (see [87]).

(2) Alonzo Church invented a formal system called the lambda calculus and defined the notion of computable function via this system, see [22].

In [88] it was proved that both models are equally strong in the sense that they define the same class of computable functions.

Based on the concept of a Turing machine are the present-day von Neumann computers. Conceptually these are Turing machines with random-access registers. Imperative programming languages such as FORTRAN, Pascal etcetera, as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements. Functional programming languages, like ML, MIRANDA, HOPE etcetera, are related to the lambda calculus. An early (although somewhat hybrid) example of such a language is LISP. Reduction machines are specifically designed for the execution of these functional languages.

Böhm and Gross [13] and Landin [58, 59, 60] were the first to suggest the importance of lambda calculus for programming. In fact, [59] already describes several aspects of functional languages developed much later. In [57] a reduction machine for lambda calculus is described. Because this resulted in rather slow actual implementations people did not believe very much in the possibility of functional programming. Wadsworth [93] introduced the notion of graph reduction that, at least in theory, could improve the performance of implementations of functional languages. Turner [90] made an implementation using graph reduction that was reasonably fast. The period 1965–1979 is a long one, but matters were going slowly because technology had provided fast and cheap von Neumann computers and little need was felt for new kinds of languages or machines. In the meantime Backus [4] had emphasised the limitations of imperative programming languages. In the early eighties things went faster and rather good sequential implementations of functional languages were developed, e.g. the G-machine [3, 49] or the compiler described in [15]. Presently, several research groups are working towards fast parallel reduction machines.

Reduction and functional programming

A functional program consists of an expression \( E \) (representing both the algorithm and the input). This expression \( E \) is subject to some rewrite rules. Reduction consists of replacing a part \( P \) of \( E \) by another expression \( P' \) according to the given rewrite rules. In schematic notation

\[
E[P] \rightarrow E[P'],
\]

provided that \( P \rightarrow P' \) is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. The expression
Thus obtained is called the normal form of $E$ and constitutes the output of the given functional program.

**Example**

$$(7 + 4) \times (8 + 5 \times 3) \rightarrow 11 \times (8 + 5 \times 3)$$

$$\rightarrow 11 \times (8 + 15)$$

$$\rightarrow 11 \times 23$$

$$\rightarrow 253.$$  

In this example the reduction rules consist of the "tables" of addition and of multiplication on the numerals (as usual, $\times$ takes priority over $\times$). Note that the "meaning" of an expression is preserved after reduction: $7 + 4$ and $11$ have the same interpretation. This feature of the evaluation of functional programs is called referential transparency.

Also symbolic computations can be done by reduction. For example,

$$\text{first-of}(\text{sort}(\text{append}(\text{"dog"}, \text{"rabbit"}) (\text{sort}(\text{"mouse"}, \text{"cat"}))))$$

$$\rightarrow \text{first-of}(\text{sort}(\text{append}(\text{"dog"}, \text{"rabbit"}) (\text{"cat"}, \text{"mouse"})))$$

$$\rightarrow \text{first-of}(\text{sort}(\text{"dog"}, \text{"rabbit", "cat", "mouse"}))$$

$$\rightarrow \text{first-of}(\text{"cat", "dog", "mouse", "rabbit"})$$

$$\rightarrow \text{"cat"}.$$  

The necessary rewrite rules for append and sort can be programmed easily by a few lines in terms of more primitive rewrite rules.

Reduction systems for functional languages usually satisfy the Church-Rosser property, which implies that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced as follows:

$$(7 + 4) \times (8 + 5 \times 3) \rightarrow (7 + 4) \times (8 + 15)$$

$$\rightarrow 11 \times (8 + 15)$$

$$\rightarrow 11 \times 23$$

$$\rightarrow 253.$$  

Or even by evaluating several expressions at the same time:

$$(7 + 4) \times (8 + 5 \times 3) \Rightarrow 11 \times (8 + 15)$$

$$\rightarrow 11 \times 23$$

$$\rightarrow 253.$$  

($\Rightarrow$ indicates that several reduction steps are done in parallel). This gives the possibility of parallel execution of functional languages.

**Functional programming and process control**

A functional program transforms data into other data according to a certain algorithm. Functional programs cannot deal with input/output, cannot switch on and off lamps depending on the value of a certain parameter; in general they cannot deal with process control. These points are sometimes held as arguments against functional programming. However a reduction machine can produce code for process control,
code that is executed by some interface. A von Neumann computer also needs interfaces for I/O and other controls. Therefore, a reduction machine with environment will consist of a pure reduction machine (dealing with algorithms that transform data) together with interfaces for process control (like I/O), see Fig. 1. This is of course a logical picture. The hardware of the two parts may be geometrically interleaved or may be even the same.

![Reduction machine and Process control](image)

**Fig. 1.**

**Process control via streams**

In spite of the previous remarks, there is an approach for functional languages that incorporates I/O. Suppose that we want to compute a function $F$ on several arguments $A_0, A_1, A_2, \ldots$ appearing consecutively in time. One can view this so-called stream $(A_0, A_1, A_2, \ldots)$ as a potentially infinite list $A = (A_0; A_1; A_2; \cdots; A_n; \perp)$ where $\perp$ stands for “unspecified” and is interactively updated to $A_{n+1}; \perp$ each time the user has a new argument $A_{n+1}$. Then on this list $A$, the system applies the function $F^*$ defined by

$$F^*(A; B) = (FA); (F*B)$$

obtaining

$$F^*A = (FA_0; FA_1; FA_2; \cdots; FA_n; F^*\perp)$$

also appearing consecutively in time.

This idea of using streams is used in some (implementations of) functional languages, e.g. in the Miranda implementation of Turner [92]. We prefer not to use this mechanism in a pure reduction machine. The reason is that, although $F^*$ is purely functional, the use of streams is not. The way $\perp$ is treated depends essentially on process control. Moreover, it is not always natural to simulate process-like actions using streams as above. For example, this is the case with “control-C”, the statement to stop a process of computation that is not yet finished. Therefore we view process control as a necessary interface.

2. **Lambda calculus**

In this section we introduce lambda calculus and show how this system is able to capture all computable functions. For more information, see [7, 42].
2.1. Conversion

We start with an informal description of the system.

Application and abstraction

The lambda calculus has two basic operations. The first one is application. The expression

\[ F \cdot A \]

(usually written as \( A \)) denotes the data \( F \) considered as an algorithm applied to \( A \) considered as input. The theory is type-free: it is allowed to consider expressions like \( FF \), that is, \( F \) applied to itself. This will be useful to simulate recursion.

The other basic operation is abstraction: if \( M = M[x] \) is an expression containing ("depending on") \( x \), then \( \lambda x.M[x] \) denotes the map

\[ x \mapsto M[x] \]

Application and abstraction work together in the following intuitive formula:

\[ (\lambda x.2\times x+1)3 = 2\times 3 + 1 \quad (= 7). \]

That is, \( \lambda x.2\times x+1 \) denotes the function \( x \mapsto 2\times x+1 \) applied to the argument \( 3 \), giving \( 2\times 3 + 1 \) (which is \( 7 \)). In general we have

\[ (\lambda x.M[x])N = M[N]. \]

This last equation is preferably written as

\[ (\beta) \quad (\lambda x.M[x])N = M[N]. \]

where \([x:=N]\) denotes substitution of \( N \) for \( x \). This equation is called \( \beta \)-conversion.

It is remarkable that although it is the only essential axiom of lambda calculus, the resulting theory is rather involved.

Free and bound variables

Abstraction is said to bind the free variable \( x \) in \( M \). For instance, we say that \( \lambda x.\cdotyx \) has \( x \) as bound and \( y \) as free variable. Substitution \([x:=N]\) is only performed in the free occurrences of \( x \):

\[ yx(\lambda x.x)[x:=N] = yN(\lambda x.x). \]

In calculus there is a similar variable binding. In \( \int_a^b f(x, y) \, dx \) the variable \( x \) is bound and \( y \) is free. It does not make sense to substitute \( 7 \) for \( x \): \( \int_a^b f(7, y) \, dy \); but substitution for \( y \) does make sense: \( \int_a^b f(x, 7) \, dx \).

For reasons of hygiene it will always be assumed that the bound variables occurring in a certain expression are different from the free ones. This can be fulfilled by renaming bound variables. For instance, \( \lambda x.x \) becomes \( \lambda y.y \). Indeed, these expressions act the same way:

\[ (\lambda x.x)a = a = (\lambda y.y)a \]

and in fact they denote the same intended algorithm. Therefore expressions that differ
only in the names of bound variables are identified. Equations like $\lambda x.x = \lambda y.y$ are usually called $\alpha$-conversion.

**Functions of several arguments**

Functions of several arguments can be obtained by iteration of application. This is due to Schönfinkel [79] but is often called "currying", after H.B. Curry who made the method popular. Intuitively, if $f(x, y)$ depends on two arguments, one can define

$$F_x = \lambda y.f(x, y), \quad F = \lambda x.F_x.$$  

Then

$$(Fx)y = F_xy = f(x, y).$$

(2.1)

This last equation shows that it is convenient to use *association to the left* for iterated application:

$$FM_1 \ldots M_n \text{ denotes } (((FM_1)M_2) \ldots M_n).$$

Equation (2.1) then becomes $Fxy = f(x, y)$.

Dually, iterated abstraction uses *association to the right*:

$$\lambda x_1 \ldots x_n.f(x_1 \ldots x_n) \text{ denotes } \lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.f(x_1 \ldots x_n))\ldots)).$$

Then we have, for $F$ defined above, $F = \lambda xy.f(x, y)$ and (2.1) becomes

$$(\lambda xy.f(x, y))xy = f(x, y).$$

For $n$ arguments we have

$$(\lambda x_1 \ldots x_n.f(x_1 \ldots x_n))x_1 \ldots x_n = f(x_1 \ldots x_n)$$

by using $n$ times (B). This last equation becomes, in convenient vector notation,

$$(\lambda x.f(x))\vec{x} = f(\vec{x});$$

more generally, one has for $\vec{N} = N_1, \ldots, N_n$

$$(\lambda x.f(x))\vec{N} = f(\vec{N}).$$

Now we give the formal description of lambda calculus.

**2.1.1. Definition.** The set of $\lambda$-terms (notation $\Lambda$) is built up from an infinite set of constants $C = \{c, c', c'', \ldots\}$ and of variables $V = \{v, v', v'', \ldots\}$ using application and (function) abstraction.

$$c \in C \Rightarrow c \in \Lambda, \quad x \in V \Rightarrow x \in \Lambda,$$

$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda, \quad M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda.$$  

In BN-form this is

constant ::= "c" | constant "'"  
variable ::= "v" | variable "'"  
$\lambda$-term ::= constant | variable | "("$\lambda$-term $\lambda$-term")" | "(\lambda"variable $\lambda$-term")".
Or, using abstract syntax, (see the chapter by Mosses in this Handbook), one may write
\[ A = C | V | AA | \lambda V A. \]

2.1.2. Example. The following are \( \lambda \)-terms:
\[ v, \quad (vc), \quad (\lambda v(vc)), \quad (v'(\lambda v(vc))), \quad (\lambda v(vc))v'. \]

2.1.3. Convention. (i) \( c, d, e, \ldots \) denote arbitrary constants, \( x, y, z, \ldots \), denote arbitrary variables, \( M, N, L, \ldots \) denote arbitrary \( \lambda \)-terms. Outermost parentheses are not written.

(ii) As already mentioned informally, the following abbreviations are used:

\[ FM_1 \ldots M_n \text{ stands for } (\ldots ((FM_1)M_2)\ldots )M_n \]

and
\[ \lambda x_1 \ldots x_n . M \text{ stands for } \lambda x_1(\lambda x_2(\ldots (\lambda x_n(M))\ldots )). \]

The examples in 2.1.2 now may be written as follows:
\[ x, \quad xc, \quad \lambda x.xc, \quad y(\lambda x.xc), \quad (\lambda x.xc)y. \]

Note that \( \lambda x . yx \) is \( (\lambda x(yx)) \) and not \( ((\lambda xy)x) \).

2.1.4. Definition. (i) The set of free variables of \( M \) (notation \( FV(M) \)) is inductively defined as follows:
\[ FV(x) = \{ x \}; \quad FV(MN) = FV(M) \cup FV(N); \]
\[ FV(\lambda x.M) = FV(M) - \{ x \}. \]

(ii) \( M \) is a closed \( \lambda \)-term (or combinator) if \( FV(M) = \emptyset \). The set of closed \( \lambda \)-terms is denoted by \( A^0 \).

In the \( \lambda \)-term \( y(\lambda xy.xyz) \), \( y \) and \( z \) occur as free variables; \( x \) and \( y \) occur as bound variables. The term \( \lambda xy.xxy \) is closed.

Now we introduce lambda calculus as a formal theory of equations between \( \lambda \)-terms.

2.1.5. Definition. (i) The principal axiom scheme of lambda calculus is

\[ (\beta) \quad (\lambda x . M) N = M[ x := N ] \quad \text{for all } M, N \in A. \]

(ii) There are also "logical" axioms and rules:
\[ M = M, \]
\[ M = N \Rightarrow N = M, \quad M = N, N = L \Rightarrow M = L, \]
\[ M = M' \Rightarrow MZ = M'Z, \quad M = M' \Rightarrow ZM = ZM', \]

(rule \( \xi \) )
\[ M = M' \Rightarrow \lambda x . M = \lambda x . M'. \]

(iii) If \( M = N \) is provable in the lambda calculus, then we write \( \lambda \vdash M = N \) or often just \( M = N \) and say that \( M \) and \( N \) are (\( \beta \))-convertible. \( M \equiv N \) denotes that \( M \) and \( N \) are the same term or can be obtained from each other by renaming bound variables. For
instance,

\((\lambda x.y)z \equiv (\lambda x.y)z,\) \hspace{1cm} \((\lambda x.x)z \equiv (\lambda y.y)z,\) \hspace{1cm} \((\lambda x.x)z \neq (\lambda x.y)z.\)

**Remark.** We have identified terms that differ only in the names of bound variables. An alternative is to add to the lambda calculus the following axiom scheme:

\[(\alpha) \quad \lambda x. M = \lambda y. M[x := y],\]

provided that \(y\) does not occur in \(M\). The axiom \(\beta\) above was originally the second axiom; hence its name. We prefer our version of the theory in which the identifications are made on a syntactic level. These identifications are done in our mind and not on paper. For implementations of the lambda calculus, the machine has to deal with this so-called \(\alpha\)-conversion. A good way of doing this is provided by the name-free notation of de Bruijn (see [7, Appendix C]). In this notation \(\lambda x.(\lambda y.x y)\) is denoted by \(\lambda(\lambda 21)\), the 2 denoting a variable bound “two lambdas above”.

**Development of the theory**

2.1.6. **Examples** (standard combinators). Define the combinators

\[
\begin{align*}
I &\equiv \lambda x.x, \\
K &\equiv \lambda x.y.x, \\
K_2 &\equiv \lambda x.y.y, \\
S &\equiv \lambda x.y.z.x(z(yz)).
\end{align*}
\]

Then the following equations are provable:

\[
\begin{align*}
IM &= M, \\
KMN &= M, \\
K_2MN &= N, \\
SMNL &= ML(NL).
\end{align*}
\]

The following result provides one way to represent recursion in the lambda-calculus.

2.1.7. **Fixed Point Theorem.** (i) \(\forall F \exists X \forall X FX = X.\) (This means that for all \(F \in A\) there is an \(X \in A\) such that \(\lambda X FX = X.\))

(ii) There is a fixed point combinator

\[
Y \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx))
\]

such that

\[
\forall F F(YF) = YF.
\]

**Proof.** (i) Define \(W \equiv \lambda x.F(xx)\) and \(X \equiv WW\). Then

\[
X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX.
\]

(ii) By the proof of (i).

2.1.8. **Application.** Given a context \(C[f, x]\) (that is, a term possibly containing the displayed free variables), then

\[
\exists F \forall X FX = C[F, X].
\]
Here $C[F, X]$ is of course the substitution result $C[f, x][f := F][x := X]$. Indeed,

$$\forall X \; FX = C[F, X] \iff Fx = C[F, x]$$

$$\iff F = \lambda x. C[F, x]$$

$$\iff F = (\lambda f x. C(f, x))F$$

$$\iff F \equiv Y(\lambda f x. C[f, x]).$$

This also holds for more arguments: $\exists F \forall x F\overline{x} = C[F, \overline{x}].$

In lambda calculus one can define numerals and represent numeric functions on them.

2.1.9. Definition. (i) $F^n(M)$ with $n \in \mathbb{N}$ (the set of natural numbers) is defined inductively as follows:

$$F^0(M) \equiv M; \quad F^{n+1}(M) \equiv F(F^n(M)).$$

(ii) The Church numerals $c_0, c_1, c_2, \ldots$ are defined by

$$c_n \equiv \lambda f x. f^n(x).$$

2.1.10. Proposition (Rosser). Define

$$A_+ \equiv \lambda x y p q. x p (y p q); \quad A_\times \equiv \lambda x y z. x (y z); \quad A_{\text{exp}} \equiv \lambda x y. y x.$$

Then one has, for all $n, m \in \mathbb{N},$

(i) $A_+ c_n c_m = c_{n+m};$

(ii) $A_\times c_n c_m = c_{n \times m};$

(iii) $A_{\text{exp}} c_n c_m = c_{(n^m)}$, except for $m = 0$ (Rosser started counting from 1).

Proof. Lemma: (i) $(c_n x)^m(y) = x^{n \times m}(y);$  
(ii) $(c_n^m(x)) = c_{(n^m)}(x)$ for $m > 0.$

Proof of lemma: (i) By induction on $m$: If $m = 0$, then LHS $= y = $ RHS. Assume (i) is correct for $m$ (induction hypothesis, abbreviated IH). Then

$$(c_n x)^{m+1}(y) = c_n x ((c_n x)^m(y)) = \text{IH} \; c_n x (x^{n \times m} y) = x^n (x^{n \times m} y)$$

$$\equiv x^{n \times m + 1}(y).$$

(ii) By induction on $m > 0$: If $m = 1$, then LHS $\equiv c_n x \equiv $ RHS. If (ii) is correct for $m$, then

$$c_n^{m+1}(x) = c_n (c_n^m(x)) = \text{IH} \; c_n (c_{(n^m)}(x)) = \lambda y. (c_{(n^m)}(x))^m(y)$$

$$= (i) \; \lambda y. x^{n \times m^n}(y) = c_{(n^{m+1})} x.$$

Now the proof of the proposition easily follows.

(i) By induction on $m$.

(ii) Use Lemma (i).

(iii) By Lemma (ii) we have, for $n > 0,$

$$A_{\text{exp}} c_n c_m = c_{m} c_n = \lambda x. c_m^n(x) = \lambda x. c_{(n^m)} x = c_{(n^m)}.$$
since $\lambda x. M x = M$ if $M \equiv \lambda y. M'[y]$. Indeed,

$$\lambda x. M x \equiv \lambda x. (\lambda y. M'[y]) x = \lambda x. M'[x] \equiv \lambda y. M'[y] \equiv M.$$  

2.2. **Representing the computable functions**

We have seen that the functions plus, times and exponentiation on $\mathbb{N}$ can be represented in the lambda calculus using Church numerals. In this section we will show that all computable (recursive) functions can be represented. In order to do this we will first introduce Booleans, pairs and a different system of numerals.

2.2.1. **Definition.** (i) $\text{true} \equiv K$, $\text{false} \equiv K\cdot$.

(ii) If $B$ is a Boolean, i.e. a term that is either $\text{true}$, or $\text{false}$, then

$$\text{if } B \text{ then } P \text{ else } Q \text{ can be represented by } B P Q.$$  

Indeed, $\text{true} P Q = K P Q = P$ and $\text{false} P Q = K \cdot P Q = Q$ (see Example 2.1.6).

2.2.2. **Definition (ordered pairs).** For $M, N \in L$ write

$$[M, N] = \lambda z. z MN.$$  

Then

$$[M, N]\text{true} = M, \quad [M, N]\text{false} = N$$  

and hence $[M, N]$ can serve as an ordered pair.

2.2.3. **Definition (numerals).** $r^0 \equiv 1, \quad r^n + 1 \equiv [\text{false}, r^n].$

2.2.4. **Lemma (successor, predecessor, test for zero).** There are combinators $S^+, P^-$, $\text{zero}$ such that for all $n \in \mathbb{N}$ one has

$$S^+ r^n = r^{n+1}, \quad P^- r^n + 1 = r^n,$$

$$\text{Zero} r^0 = \text{true}, \quad \text{Zero} r^n + 1 = \text{false}.$$  

**Proof.** Take

$$S^+ \equiv \lambda x.[\text{false}, x], \quad P^- \equiv \lambda x.x\text{false}, \quad \text{Zero} \equiv \lambda x.x\text{true}.$$  

2.2.5. **Definition.** (i) A numeric function is a map $f: \mathbb{N}^p \to \mathbb{N}$ for some $p$.

(ii) A numeric function $f$ with $p$ arguments is called $\lambda$-definable if one has for some combinator $F$

$$F \left( n_1, \ldots, n_p \right) = f(n_1, \ldots, n_p)$$  

(2.2) for all $n_1, \ldots, n_p \in \mathbb{N}$. If (2.2) holds, then $f$ is said to be $\lambda$-defined by $F$.

2.2.6. **Definition.** (i) The initial functions are the numeric functions $U_n^i, S^+, Z$ defined
by
\[U'_i(x_1, \ldots, x_n) = x_i, \quad 1 \leq i \leq n,\]
\[S^+(n) = n + 1, \quad Z(n) = 0.\]

(ii) Let \( P(n) \) be a numeric relation. As usual, \( \mu m [P(m)] \) denotes the least number \( m \) such that \( P(m) \) holds if there is such a number; otherwise it is undefined.

2.2.7. Definition. Let \( \mathcal{A} \) be a class of numeric functions.
(i) \( \mathcal{A} \) is closed under composition if, for all \( f \) defined by
\[f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_m(\bar{n})) \quad \text{with} \quad g, h_1, \ldots, h_m \in \mathcal{A},\]
one has \( f \in \mathcal{A} \).
(ii) \( \mathcal{A} \) is closed under primitive recursion if, for all \( f \) defined by
\[f(0, \bar{n}) = g(\bar{n}), \quad f(k + 1, \bar{n}) = h(f(k, \bar{n}), k, \bar{n}) \quad \text{with} \quad g, h \in \mathcal{A},\]
one has \( f \in \mathcal{A} \).
(iii) \( \mathcal{A} \) is closed under minimalization if, for all \( f \) defined by
\[f(\bar{n}) = \mu m [g(\bar{n}, m) = 0] \quad \text{with} \quad g \in \mathcal{A} \quad \text{such that} \quad \forall \bar{n} \exists m g(\bar{n}, m) = 0,\]
one has \( f \in \mathcal{A} \).

2.2.8. Definition. The class \( \mathcal{R} \) of recursive functions is the smallest class of numeric functions that contains all initial functions and is closed under composition, primitive recursion and minimalization.

So \( \mathcal{R} \) is an inductively defined class. The proof that all recursive functions are \( \lambda \)-definable is by a corresponding induction argument. The result is originally due to Kleene [54].

2.2.9. Lemma. The initial functions are \( \lambda \)-definable.

Proof. Take as defining terms
\[U_p^i \equiv \lambda x_1 \ldots x_p x_i, \quad S^+ \equiv \lambda x. [ \text{false}, x ], \quad Z \equiv \lambda x. \cdot 0^\ast. \]

2.2.10. Lemma. The \( \lambda \)-definable functions are closed under composition.

Proof. Let \( g, h_1, \ldots, h_m \) be \( \lambda \)-defined by \( G, H_1, \ldots, H_m \) respectively. Then \( f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_m(\bar{n})) \) is \( \lambda \)-defined by
\[F \equiv \lambda \bar{x}. G(H_1 \bar{x}) \ldots (H_m \bar{x}). \]

2.2.11. Lemma. The \( \lambda \)-definable functions are closed under primitive recursion.

Proof. Let \( f \) be defined by
\[f(0, \bar{n}) = g(\bar{n}), \quad f(k + 1, \bar{n}) = h(f(k, \bar{n}), k, \bar{n})\]
where \( g, h \) are \( \lambda \)-defined by \( G, H \) respectively. An intuitive way to compute \( f(k, \bar{n}) \) is the following:

- test whether \( k = 0 \);
- if yes, then give output \( g(\bar{n}) \);
- if no, then compute \( h(f(k-1, \bar{n}), k-1, \bar{n}) \).

Therefore we want a term \( F \) such that

\[
F\bar{x}\bar{y} = \text{if } (\text{Zero } x) \text{ then } G\bar{y} \text{ else } H(F(\text{P}^-x)\bar{y})(\text{P}^-x)\bar{y} \\
\equiv D(F, x, \bar{y}).
\]

Now such an \( F \) can be found by Application 2.1.8 of the Fixed Point Theorem. \( \square \)

**2.2.12. Lemma.** The \( \lambda \)-definable functions are closed under minimalization.

**Proof.** Let \( f \) be defined by \( f(\bar{n}) = \mu m [g(\bar{n}, m) = 0] \), where \( g \) is \( \lambda \)-defined by \( G \). Again, by the Fixed Point Theorem, there is a term \( H \) such that

\[
H\bar{x}\bar{y} = \text{if } (\text{zero}(G\bar{x}\bar{y})) \text{ then } y \text{ else } H\bar{x}(S^+y).
\]

Set \( F = \lambda \bar{x}.H\bar{x}[0] \). Then \( F \) \( \lambda \)-defines \( f \):

\[
F[\bar{n}] = H\bar{y}\bar{n}^+r0^-
\]

\[
= \begin{cases} 
0^+ & \text{if } G\bar{y}\bar{n}^+r0^+ = 0^+ \\
H\bar{y}\bar{n}^+r1^- & \text{else} \\
1^+ & \text{if } G\bar{y}\bar{n}^+r1^- = 0^+ \\
H\bar{y}\bar{n}^+r2^- & \text{else} \\
2^+ & \text{if } \ldots \\
\ldots & \text{else}
\end{cases}
\]

**2.2.13. Theorem.** All recursive functions are \( \lambda \)-definable.

**Proof.** By Lemmas 2.2.9–12. \( \square \)

The converse also holds. The idea is that if a function is \( \lambda \)-definable, then its graph is recursively enumerable, because equations derivable in the lambda calculus can be enumerated. It then follows that the function is recursive. So, for numeric functions, we have that \( f \) is recursive iff \( f \) is \( \lambda \)-definable. Moreover, also for partial functions, a notion of \( \lambda \)-definability exists. If \( \psi \) is a partial numeric function, then we have

\( \psi \) is partial recursive iff \( \psi \) is \( \lambda \)-definable.

The notions \( \lambda \)-definable and recursive both are intended to be formalisations of the intuitive concept of computability. Another formalisation was proposed by Turing in the form of Turing computability. The equivalence of the notions recursive, \( \lambda \)-definable and Turing computable (see, besides the original [88], also [30]) provides evidence for the Church–Turing thesis that states that "recursive" is the proper formalisation of the intuitive notion "computable".
Now we show the $\lambda$-definability of recursive functions with respect to the Church numerals.

2.2.14. **Theorem.** With respect to the Church numerals $c_n$ all recursive functions can be $\lambda$-defined.

**Proof.** Define

$$S^+_c \equiv \lambda xyz. y(xyz),$$
$$P^+_c \equiv \lambda xyz. x(\lambda pq.q(py))(Kz)l \quad \text{(this term was found by J. Velmans)},$$
$$\text{zero}_c \equiv \lambda x. xS^+r0^3\text{true}.$$

Then these terms represent the successor, predecessor and test for zero. Then, as before, all recursive functions can be $\lambda$-defined. □

An alternative proof uses "translators" between the numerals $^r+n^3$ and $c_n$.

2.2.15. **Proposition.** There exist terms $T$ and $T^{-1}$ such that for all $n$

(i) $Tc_n = ^r+n^3$;

(ii) $T^{-1}r0^3 = c_n$.

**Proof.** (i) Take $T \equiv \lambda x. xS^+r0^3$.

(ii) Take $T^{-1} \equiv \lambda x. \text{if Zero } x \text{ then } c_0 \text{ else } S^+_c(T^{-1}(P^{-1}x)). \quad □$

2.2.16. **Corollary.** Second proof of Theorem 2.2.14.

**Proof.** Let $f$ be a recursive function (of arity 2, say). Let $F$ represent $f$ with respect to the numerals $^r+n^3$. Define

$$F_c = \lambda xy. T^{-1}(F(Tx)(Ty)).$$

Then $F_c$ represents $f$ with respect to the Church numerals. □

We end this section with some results showing the flexibility of lambda calculus. First, we give the double Fixed Point Theorem.

2.2.17. **Theorem.** $\forall A, B \exists X, Y \ X = AX Y \ & \ Y = BX Y$.

**Proof.** Define $F \equiv \lambda x. [A(x\text{true})(x\text{false}), B(x\text{true})(x\text{false})]$. By the simple Fixed Point Theorem 2.1.7 there exists a $Z$ such that $FZ = Z$. Take $X = Z\text{true}$, $Y = Z\text{false}$. Then

$$X = Z\text{true} = FZ\text{true} = A(Z\text{true})(Z\text{false}) \equiv AX Y$$

and, similarly, $Y = BX Y$. □

**Alternative Proof** (Smullyan). By the ordinary Fixed Point Theorem, we can
construct a term $N$ such that

$$Nxyz = x(Nyyz)(Nzyz).$$

Now take $X = NAAB$ and $Y = NBAB$. □

2.2.18. Application. Given context $C_i \equiv C_i[f, g, \overline{x}]$, $i = 1, 2$, there exist $F_1, F_2$ such that

$$F_1\overline{x} = C_1[F_1, F_2, \overline{x}] \quad \text{and} \quad F_2\overline{x} = C_2[F_1, F_2, \overline{x}].$$

Proof. Define $A_i \equiv \lambda f_1 f_2 x C_i[f_1, f_2, \overline{x}]$ and apply Theorem 2.2.17 to $A_1, A_2$. □

Of course, Theorem 2.2.17 (both proofs) and Application 2.2.18 generalise to a $k$-fold fixed point theorem and $k$-fold recursion.

Now it will be shown that there is a "self-interpreter" in the lambda calculus, i.e. a term $E$ such that for closed terms $M$ (without constants) we have the $E$ applied to the "code" of $M$ yields $M$ itself (which is an executable on a reduction machine). First, we give a definition of coding.

2.2.19. Definition (coding $\lambda$-terms without constants)

(i) $v^{(0)} = v$; $v^{(n+1)} = v^{(n)}$; similarly, one defines $c^{(n)}$.

(ii) $\# v^{(n)} = <0, n>$; $\# c^{(n)} = <1, n>$;

$$\# (M N) = <2, \# M, \# N>$$

$$(\lambda x.M) = <3, \# x, \# M>.$$ 

(iii) Notation: $r^\gamma M^n = r^\gamma \# M^n$.

2.2.20. Theorem (Kleene). There is an "interpreter" $E \in \Lambda^0$ for closed $\lambda$-terms without constants. That is, for all closed $M$ without constants one has $E^\gamma M^n = M$.

Proof (P. de Bruin). Construct an $E_1$ such that

$$E_1 F^\gamma x^n = F^\gamma x^n,$$

$$E_1 F^\gamma M N^n = (E_1 F^\gamma M^n) (E_1 F^\gamma N^n),$$

$$E_1 F^\gamma \lambda x.M^n = \lambda z.(E_1 F_{[x=z]}^\gamma M^n)$$

where

$$F_{[x=z]}^\gamma n^n = \begin{cases} F^\gamma n^n & \text{if } n \neq \# x, \\ z & \text{if } n = \# x. \end{cases}$$

(Note that $F_{[x=z]}^\gamma$ can be written in the form $G^\gamma x^n z F$.) By induction on $M$ if follows that

$$E_1 F^\gamma M^n = M[x_1 := F^\gamma x_1^n, \ldots, x_n := F^\gamma x_n^n][c_1 := F^\gamma c_1^n, \ldots, c_m := F^\gamma c_m^n]$$

where $\{x_1, \ldots, x_n\} = FV(M)$ and $\{c_1, \ldots, c_m\}$ are the constants in $M$. Hence, for closed $M$ without constants, one has $E_1^\gamma M^n = M$. Now take $E \equiv \lambda x. E_1^\gamma 1 x = E_1^\gamma 1$. □

For terms possibly containing a fixed finite set of constants $\overline{c}$ and variables $\overline{x}$, one can also construct an interpreter $E_{\overline{c}, \overline{x}}$. 
2.2.21. APPLICATION. Remember \( U_n^m = \lambda x_1 \ldots x_n. x_n \). Construct an \( F \in \mathcal{A}^0 \) such that \( F^r M^n = U_n^m \).

Solution. Clearly, \( \# U_n^m \) depends recursively on \( n \). Therefore, for some recursive \( g \) with \( \lambda \)-defining term \( G \), one has
\[
\# U_n^m = g(n) \Rightarrow \# U_n^m = G^r n^n.
\]
Now take \( F = \lambda x. E(Gx) \). Then
\[
F^r n^n = E(G^r n^n) = E^r U_n^m = U_n^m.
\]

Exercise (Vree). Find a relatively simple \( F \) such that \( F^r n^n = U_n^m \).

Now we prove the so-called second Fixed Point Theorem.

2.2.22. Theorem. \( \forall F \exists X F^r X = X \).

Proof. By the effectiveness of \( \# \), there are recursive functions Ap and Num such that \( \text{Ap}(\# M, \# N) = \# MN \) and \( \text{Num}(n) = \# n^n \). Let Ap and Num be \( \lambda \)-defined by \( \text{Ap} \) and \( \text{Num} \in \mathcal{A}^0 \). Then
\[
\begin{align*}
\text{Ap}^r M^n M^N &= M N^n, \\
\text{Num}^r n^n &= n^n;
\end{align*}
\]
hence, in particular, \( \text{Num}^r M^n = M^r n^n \). Now define
\[
W = \lambda x. F(\text{Ap}(\text{Num}(x))), \quad X = W^r W^n.
\]
Then
\[
\begin{align*}
X &= W^r W^n = F(\text{Ap}^r W^n(\text{Num}^r W^n)) \\
&= F^r W^r W^n = F^r X^n.
\end{align*}
\]

An application will be given in Section 5.2. Another application is the following result, due to Scott, which is quite useful for proving undecidability results.

2.2.23. Theorem. Let \( A \subseteq \mathcal{A} \) such that \( A \) is nontrivial, i.e. \( A \neq \emptyset, A \neq \mathcal{A} \). Suppose that \( A \) is closed under \( \subseteq \), that is,
\[
M \in A, M = N \Rightarrow N \in A.
\]
Then \( A \) is not recursive; that is \( \# A = \{ \# M \mid M \in A \} \) is not recursive.

Proof. Suppose \( A \) is recursive. It follows that there is an \( F \in \mathcal{A}^0 \) with
\[
M \in A \iff F^r M^n = 0^n, \quad M \notin A \iff F^r M^n = 1^n.
\]
Let \( M_0 \in A, M_1 \notin A \). We can find a \( G \in \mathcal{A} \) such that
\[
M \in A \iff G^r M^n = M_1 \notin A, \quad M \notin A \iff G^r M^n = M_0 \in A.
\]
(Take \( Gx = \text{if Zero}(Fx) \text{ then } M_1 \text{ else } M_0. \)) By the second Fixed Point Theorem, there is
a term $M$ such that $G^\gamma M^\gamma = M$. Hence,
$$M \in A \iff M = G^\gamma M^\gamma \notin A,$$
a contradiction. □

The following application, first proved in [22] using another method, was in fact historically the first example of a noncomputable property. We say that a term $M$ is in normal form (nf) if $M$ has no part of the form $(\lambda x.P)Q$. A term $M$ has a normal form if $M = M'$ and $M'$ is in nf. For example $1$ is in nf and $1K$ has a nf.

2.2.24. Corollary. The set $NF = \{M | M$ has a nf$\}$ is not recursive.

Proof. $NF$ is closed under $=$ and is nontrivial. □

3. Semantics

Semantics of a (formal) language $L$ gives a “meaning” to the expressions in $L$. This meaning can be given in two ways: (1) By providing a way in which expressions of $L$ are used. (2) By translating the expressions of $L$ into expressions of another language that is already known.

In Section 3.1 the meaning of a $\lambda$-term is explained through the notions reduction and strategy. Once a reduction strategy is chosen, the behaviour of a term is determined. This gives a so-called operational semantics.

In Section 3.2 the meaning of a $\lambda$-term is given by translating it to (an expression denoting) a set $[M]$. This set is an element of a mathematical structure in which application and abstraction are well-defined operations and the map $[\ ]$ preserves these operations. In this way we obtain a so-called denotational semantics.

In both cases the semantics gives a particular view on terms and classifies them. For example, in the operational semantics two terms may act the same on the numerals. The equivalence relation on terms induced by a denotational semantics, i.e. $M \approx N$ iff $[M] = [N]$, is often of considerable interest and can settle questions of purely syntactical character; see for example the proof of Theorem 21.2.21 in [7].

3.1. Operational semantics: reduction and strategies

There is a certain asymmetry in the basic scheme (β). The statement
$$(\lambda x.x^2 + 1) \ 3 = 3^2 + 1$$
can be interpreted as “$3^2 + 1$ is the result of computing $(\lambda x.x^2 + 1) \ 3$”, but not vice versa. This computational aspect will be expressed by writing
$$(\lambda x.x^2 + 1) \ 3 \triangleright\triangleright 3^2 + 1$$
which reads “$(\lambda x.x^2 + 1) \ 3$ reduces to $3^2 + 1$”.

Apart from this conceptual aspect, reduction is also useful for an analysis of
convertibility. The Church–Rosser Theorem says that if two terms are convertible, then there is a term to which they both reduce. In many cases the inconvertibility of two terms can be proved by showing that they do not reduce to a common term.

3.1.1. Definition. (i) A binary relation \( R \) on \( A \) is called compatible (with the operations) if
\[
M R N \Rightarrow ZM R ZN, MZ R NZ \text{ and } \lambda x.M R \lambda x.N.
\]
(ii) A congruence relation on \( A \) is a compatible equivalence relation.
(iii) A reduction relation on \( A \) is a compatible reflexive and transitive relation.

3.1.2. Definition. The binary relations \( \rightarrow_{\beta}, \rightarrow_{\beta}^{*} \) and \( =_{\beta} \) on \( A \) are defined inductively as follows:

(i) (1) \( (\lambda x.M)N \rightarrow_{\beta} M[x:=N] \);
     (2) \( M \rightarrow_{\beta} N \Rightarrow ZM \rightarrow_{\beta} ZN, \)
     \( M \rightarrow_{\beta} N \Rightarrow MZ \rightarrow_{\beta} NZ, \)
     \( M \rightarrow_{\beta} N \Rightarrow \lambda x.M \rightarrow_{\beta} \lambda x.M. \)

(ii) (1) \( M \rightarrow_{\beta}^{*} M; \)
     (2) \( M \rightarrow_{\beta} N \Rightarrow M \rightarrow_{\beta}^{*} N; \)
     (3) \( M \rightarrow_{\beta}^{*} N, N \rightarrow_{\beta}^{*} L \Rightarrow M \rightarrow_{\beta}^{*} L. \)

(iii) (1) \( M \rightarrow_{\beta} N \Rightarrow M =_{\beta} N; \)
     (2) \( M =_{\beta} N \Rightarrow N =_{\beta} M; \)
     (3) \( M =_{\beta} N, N =_{\beta} L \Rightarrow M =_{\beta} L. \)

These relations are pronounced as follows:
• \( M \rightarrow_{\beta} N \) reads "\( M \) \( \beta \)-reduces to \( N \"; 
• \( M \rightarrow_{\beta}^{*} N \) reads "\( M \) \( \beta \)-reduces to \( N \) in one step"; 
• \( M =_{\beta} N \) reads "\( M \) is \( \beta \)-convertible to \( N \". 

By definition we have that \( \rightarrow_{\beta} \) is compatible, \( \rightarrow_{\beta}^{*} \) is a reduction relation and \( =_{\beta} \) is a congruence relation; \( \rightarrow_{\beta}^{*} \) is the transitive reflexive closure of \( \rightarrow_{\beta} \) and \( =_{\beta} \) is the equivalence relation generated by \( \rightarrow_{\beta} \).

3.1.3. Proposition. \( M =_{\beta} N \iff \lambda x.M = N. \)

Proof. (\( \Rightarrow \)): By induction on the generation of \( =_{\beta}. \)
(\( \Leftarrow \)): By induction on the length of proof. \( \square \)

3.1.4. Definition. (i) A \( \beta \)-redex is a term of the form \( (\lambda x.M)N \) and \( M[x:=N] \) is its contractum.
(ii) A \( \lambda \)-term \( M \) is a \( \beta \)-normal form (\( \beta \)-nf) if it does not have a \( \beta \)-redex as subexpression.
(iii) A term \( M \) has a \( \beta \)-normal form if \( M =_{\beta} N \) and \( N \) is a \( \beta \)-nf, for some \( N \).
An immediate property of nf's is the following.

3.1.5. Lemma. Let $M$ be a $\beta$-nf. Then

$M \rightarrow^\beta N \Rightarrow N \equiv M.$

Proof. This is true if $\rightarrow^\beta$ is $\rightarrow_\beta$. The result follows by transitivity. □

3.1.6. Theorem (Church-Rosser). If $M \rightarrow^\beta N_1$, $M \rightarrow^\beta N_2$, then for some $N_3$ one has $N_1 \rightarrow^\beta M_3$ and $N_2 \rightarrow^\beta N_3$ (see Fig. 2).

![Fig. 2.](image)

The proof will not be given, but can be found in, e.g., [7, Section 11.1].

3.1.7. Corollary. If $M =_\beta N$, then there is an $L$ such that $M \rightarrow^\beta L$ and $N \rightarrow^\beta L$.

Proof. Induction on the generation of $=_\beta$.

Case 1: $M =_\beta N$ because $M \rightarrow^\beta N$. Take $L \equiv N$.

Case 2: $M =_\beta N$ because $N =_\beta M$. By the IH there is a common $\beta$-reduct $L_1$ of $N$ and $M$. Take $L \equiv L_1$.

Case 3: $M =_\beta N$ because $M =_\beta N'$, $N' =_\beta N$. Apply the IH to find $L_1, L_2$ as in Fig. 3 and then use Theorem 3.1.6 to find $L$. □

![Fig. 3.](image)
3.1.8. **Corollary.** (i) If $M$ has $N$ as $\beta$-nf, then $M \rightarrow^\beta N$.
   (ii) A $\lambda$-term has at most one $\beta$-nf.

**Proof.** (i) Suppose $M =^\beta N$ with $N$ in $\beta$-nf. By Corollary 3.1.7, $M \rightarrow^\beta L, N \rightarrow^\beta L$ for some $L$. But then $N \equiv L$ by Lemma 3.1.5, so $M =^\beta N$.

(ii) Suppose $M$ has $\beta$-nf's $N_1, N_2$. Then $N_1 =^\beta N_2(=^\beta M)$. By Corollary 3.1.7, $N_1 \rightarrow^\beta L, N_2 \rightarrow^\beta L$ for some $L$. But then $N_1 \equiv L \equiv N_2$ by Lemma 3.1.5. □

Some consequences are the following:

1. The $\lambda$-calculus is consistent, i.e. $\lambda \not\vdash \text{true} = \text{false}$. Otherwise $\text{true} =^\beta \text{false}$ by Proposition 3.1.3, which is impossible by Corollary 3.1.8 since $\text{true}$ and $\text{false}$ are distinct $\beta$-nf's. This is a syntactic consistency proof.

2. $\Omega = (\lambda x.xx)(\lambda x.xx)$ has no $\beta$-nf. Otherwise $\Omega \rightarrow^\beta N$ with $N$ in $\beta$-nf. But $\Omega$ only reduces to itself and is not in $\beta$-nf.

3. In order to find the $\beta$-nf of a term $M$ (if it exists), the various subexpressions of $M$ may be reduced in different orders. By Corollary 3.1.8(ii) the $\beta$-nf is unique.

The combinator $Y$ introduced in Theorem 2.1.7. finds fixed points: $YF = F(YF)$. One does not have

$$YF \rightarrow F(YF),$$

although this is often desirable. Turing introduced another fixed point operator that does have the desired property.

3.1.9. **Theorem** (Turings fixed point combinator). Let $\Theta = AA$, with $A = \lambda xy.y(xx)$. Then $\Theta F \rightarrow F(\Theta F)$.

**Proof.** $\Theta F = AAF \rightarrow F(AAF) \equiv F(\Theta F)$. □

Similarly, one can find solutions for the double and for the second fixed point theorem that do reduce in an analogous manner.

**Strategies**

There are terms having a nf but that are such that not all reduction paths lead to the nf. For example $A \equiv KIB$, with $B$ a term without a nf, has as normal form $I$; but $A$ also has an infinite reduction path (by reducing within $B$).

A reduction strategy chooses among the various possible redexes which one(s) to contract, and thereby it determines how to reduce a term. The following theorem states that the leftmost (or also called normal) reduction strategy always normalises terms that do have a nf.

3.1.10. **Definition.** (i) The main symbol of a redex $(\lambda x.M)N$ is the first $\lambda$.

(ii) Let $R_1, R_2$ be redex occurrences in a term $M$. Then $R_1$ is to the left of $R_2$ if the main symbol of $R_1$ is to the left of that of $R_2$.

(iii) We write $M \rightarrow_r N$ if $N$ results from $N$ by contracting the leftmost redex in $M$. $\rightarrow_r$ is the transitive reflexive closure of $\rightarrow_r$. 

The following theorem, due to Curry, states that if a term has a nf, then that nf can be found by leftmost reduction. For a proof see [7, Theorem 13.2.2]. A more direct and simpler proof is in [86, Theorem 4.7].

3.1.11. Theorem. If M has a nf N, then M →_l N.

The leftmost reduction strategy is sometimes called lazy reduction, because in an expression like ((λab. C[a, b])AB) it does not first evaluate A and B to nf, but substitutes these subterms directly into C[a, b]. Eager reduction is such that first A and B are reduced to nf before these are substituted in C.

For the lambda calculus it is not possible to have an eager evaluation mechanism. (For this reason the lambda calculus is sometimes called a lazy language.) This is due to the possibility of so-called nonstrict functions like λx.[0]. It is an advantage for the expressive power of the lambda calculus, but a complication when implementing the language.

3.2. Denotational semantics: lambda models

Trying to construct a model for the lambda calculus one would like a space D such that D is isomorphic to D^D. For cardinality reasons this is impossible. In 1969 Scott solved this problem by restricting D^D to the continuous functions on D provided with a proper topology. Because of Schönfinkel's identification of D^D^D with (D^D)^D, it is natural to use a class of topological spaces that form a Cartesian closed category. For this reason Scott worked with complete lattices with an induced topology and constructed a D such that D^D ≅ D in this category. It turned out that for a model of the lambda calculus it is sufficient to find a D such that D^D is a retract of D. Such a model will be constructed in this subsection.

After Scott introduced the first set-theoretic models of the lambda calculus, various categories of domains have been studied in which these models "live". For a survey of this domain theory, see the chapter of Gunter and Scott in this Handbook.

3.2.1. Definition. A complete lattice is a partially ordered set D = (D, ≤) such that for all X ⊆ D the supremum sup X ∈ D exists.

D, D', ..., range over complete lattices. Each D has a largest element top ⊤ = sup D and a least element bottom ⊥ = sup ∅ and every X ⊆ D has an infimum inf X = sup {y | ∀x ∈ X y ≤ x}. A subset X ⊆ D is directed if X ≠ ∅ and ∀x, y ∈ X ∃z ∈ X [x ≤ z and y ≤ z].

3.2.2. Definition. A map f: D → D' is continuous if for all directed X ⊆ D one has f(sup X) = sup f(X) (= sup {f(x) | x ∈ X}).

There is a topology on complete lattices such that "continuous" in Definition 3.2.2 coincides with the usual notion.
Note that a continuous function is automatically monotonic.
\[ x \leq y \implies y = \sup \{x, y\} \]
\[ \implies f(y) = f(\sup \{x, y\}) = \sup \{f(x), f(y)\} \]
\[ \implies f(x) \leq f(y). \]

3.2.3. Definition. Let \( D = (D, \sqsubseteq), D' = (D', \sqsubseteq'). \)
(i) \( D \times D' = \{(d, d') | d \in D, d' \in D'\} \) is the Cartesian product of \( D, D' \) and is partially ordered by
\[ (d_1, d_1') \sqsubseteq (d_2, d_2') \iff d_1 \sqsubseteq d_2 \text{ and } d_1' \sqsubseteq d_2'. \]
(ii) \( [D \to D'] = \{f : D \to D' | f \text{ is continuous}\} \) is a function space partially ordered by
\[ f \sqsubseteq g \iff \forall d \in D \ f(d) \sqsubseteq g(d). \]

3.2.4. Proposition. (i) \( D \times D' \) is a complete lattice with for \( X \subseteq D \times D' \)
\[ \sup X = (\sup(X)_0, \sup(X)_1), \]
where
\[ (X)_0 = \{d \in D | \exists d' \in D'(d, d') \in X\}, \]
\[ (X)_1 = \{d' \in D' | \exists d \in D(d, d') \in X\}. \]
(ii) Let \( f_i : D \to D', i \in I, \) be a collection of continuous maps. Define \( f(x) = \sup_i (f_i(x)) \). Then \( f \) is continuous and in \( [D \to D'] \) one has \( f = \sup f_i \). Therefore \( [D \to D'] \) is a complete lattice.

Proof. (i) Easy.
(ii) Let \( X \subseteq D \) be directed. Then
\[ f(\sup X) = \sup_i f_i(\sup X) \]
\[ = \sup_i \sup_{x \in X} f_i(x) \text{ by continuity of } f_i, \]
\[ = \sup_{x \in X} \sup_i f_i(x) \]
\[ = \sup_{x \in X} f(x). \]
Therefore \( f \) is continuous. Hence \( f = \sup f_i \) in \( [D \to D'] \). \( \Box \)

If \( \lambda x \) denotes \( \lambda \)-abstraction in set theory, then we have, as a consequence of Proposition 3.2.4(ii)
\[ \sup_i \lambda x . f_i(x) = \lambda x . \sup_i (f_i(x)), \]
i.e., \( \sup \) commutes with \( \lambda \).
3.2.5. Proposition. Let \( f \in [D \to D] \). Then \( f \) has a least fixed point defined by
\[
\text{Fix}(f) = \sup_n f^n(\bot).
\]

Proof. Note that the set \( \{ f^n(\bot) \mid n \in \mathbb{N} \} \) is directed: \( \bot \subseteq f(\bot) \); hence, by monotonicity, \( f(\bot) \subseteq f^2(\bot) \) etcetera, so \( \bot \subseteq f(\bot) \subseteq f^2(\bot) \subseteq \cdots \). Therefore
\[
f(\text{Fix}(f)) = \sup_n f(f^n(\bot)) = \sup_n f^{n+1}(\bot) = \text{Fix}(f).
\]
If \( x \) is another fixed point of \( f \), then \( f(x) = x \) and \( \bot \subseteq x \); so, by monotonicity,
\[
f^n(\bot) \subseteq f^n(x) = x.
\]
Therefore \( \text{Fix}(f) \subseteq x \). □

3.2.6. Lemma. Let \( f : D \times D' \to D'' \). Then \( f \) is continuous iff \( f \) is continuous in each of its variables separately (i.e. \( \forall x. f(x, x'_0) \) and \( \forall x'. f(x_0, x') \) are continuous for all \( x_0, x'_0 \)).

Proof. (\( \Rightarrow \)): As usual.
(\( \Leftarrow \)): Let \( X \subseteq D \times D' \) be directed. Then
\[
f(\sup X) = f(\sup(X)_0, \sup(X)_1)
\]
\[
= \sup_{x \in (X)_0} f(x, \sup(X)_1)
\]
\[
= \sup_{x \in (X)_0} \sup_{x' \in (X)_1} f(x, x')
\]
\[
= \sup_{(x, x') \in X} f(x, x').
\]
The last equality holds because \( X \) is directed. Therefore \( f \) is continuous. □

3.2.7. Proposition (continuity of application). Define \( \text{Ap} : [D \to D'] \times D \to D' \) by \( \text{Ap}(f, x) = f(x) \). Then \( \text{Ap} \) is continuous.

Proof. Apply Lemma 3.2.6: \( \forall x. \text{Ap}(f, x) = \text{Ap}(f, x'_0) \) is continuous since \( f \in [D \to D'] \). Let \( H = \forall f. \text{Ap}(f, x_0) = \forall f. f(x_0) \). Then, for \( f, i \in I, \) directed,
\[
H\left(\sum_i f_i\right) = \left(\sum_i f_i\right)(x_0)
\]
\[
= \sup_i (f_i(x_0)) \quad \text{by Proposition 3.2.4(ii)},
\]
\[
= \sup_i H(f_i). \quad \square
\]

3.2.8. Proposition (continuity of abstraction). Let \( f \in [D \times D' \to D''] \). Then \( \forall y. f(x, y) \in [D' \to D''] \) and depends continuously on \( x \).

Proof. By Lemma 3.2.6 it follows that \( \forall y. f(x, y) \in [D' \to D''] \). Moreover, let \( X \subseteq D \)
be directed. Then
\[ \lambda y. f(\sup X, y) = \sup_x \lambda y. f(x, y) \]
\[ = \sup_x \lambda y. f(x, y) \]
by continuity of \( f \) and the remark following Proposition 3.2.4. □

It now follows that the category of complete lattices with continuous maps forms a Cartesian closed category. We will not use this terminology however.

3.2.9. Definition. (i) \( D \) is a retract of \( D' \) (notation \( D < D' \)) if there are continuous maps \( F: D' \to D, G: D \to D' \) such that \( F \circ G = \text{id}_D \).
(ii) \( D \) is called reflexive if \([D \to D] < D\).

Remark. If \( D < D' \) via the maps \( F, G \), then \( F \) is surjective and \( G \) injective. We may identify \( D \) with its image \( G(D) \subseteq D' \). Then \( F \) “retracts” the larger space \( D' \) to the subspace \( D \).

Now it will be shown how a reflexive \( D \) can be turned into a model of the lambda-calculus.

3.2.10. Definition. Let \( D \) be reflexive via \( F, G \).
(i) \( F \) retracts \( D \) to its function space \([D \to D] \subseteq D\). So for \( x \in D \) one has \( F(x) \in [D \to D] \).
In this way elements of \( D \) become functions on \( D \) and one may write
\[ x \cdot_F y = F(x)(y) (\in D). \]
(ii) Conversely, every continuous function on \( D \) becomes via \( G \) an element of \( D \).
Now one may write
\[ \lambda^G x. f(x) = G(f)(\in D). \]
for \( f \) continuous.
A valuation in \( D \) is a map \( \rho \): variables \( \to D \).

3.2.11. Definition. Let \( D \) be reflexive via \( F, G \).
(i) Given a valuation \( \rho \) in \( D \) and \( M \in \Lambda \) the interpretation of \( M \) in \( D \) under the valuation \( \rho \) (notation \( \llbracket M \rrbracket^\rho_D \)) is defined as shown in Table 1, where \( \rho(x:=d) \) is the valuation \( \rho' \) with
\[ \rho'(y) = \begin{cases} 
\rho(y) & \text{if } y \neq x \\
 \rho(x) & \text{if } y = x.
\end{cases} \]
This definition is correct: by induction on \( P \) one can show the continuity of \( \lambda d. [P]_{\rho(x:=d)} \).
(ii) \( M = N \) is true in \( D \) (notation \( D \models M = N \)) if for all \( \rho \) one has \( \llbracket M \rrbracket^\rho_D = \llbracket N \rrbracket^\rho_D \).
Intuitively \( \llbracket M \rrbracket^\rho_D \) is \( M \) interpreted in \( D \) where each lambda calculus application .
is interpreted as \( F \) and each \( \lambda \) as \( \lambda^G \). For instance,
\[
\left[ \lambda x. xy \right]_\rho^D = \lambda^G x. x \rho(y).
\]

**Informal Notation.** If a reflexive \( D \) is given and \( \rho(y) = d \), then we will loosely write \( \lambda x. xd \) to denote the more formal \( \left[ \lambda x. xy \right]_\rho^D \).

Clearly \( \left[ M \right]_\rho^D \) depends only on the values of \( \rho \) on \( \text{FV}(M) \). That is,
\[
\rho \upharpoonleft \text{FV}(M) = \rho \upharpoonleft \text{FV}(M) \Rightarrow \left[ M \right]_\rho^D = \left[ M \right]_{\rho'}^D,
\]
where \( \upharpoonleft \) denotes function restriction. In particular for combinators, \( \left[ M \right]_\rho^D \) does not depend on \( \rho \) and may be written \( \left[ M \right]^D \). If \( D \) is clear from the context we write \( \left[ M \right]_{\rho} \) or \( \left[ M \right] \).

**3.2.12. Theorem.** If \( D \) is reflexive, then \( D \) is a sound model for the lambda calculus, i.e.
\[
\lambda \vdash M = N \Rightarrow D \vdash M = N.
\]

**Proof.** Induction on the proof of \( M = N \). The only two interesting cases are the axioms (\( \beta \)) and the rule (\( \xi \)).

As to (\( \beta \)). This was the scheme \( (\lambda x. M) N = M \downarrow \left[ x := N \right] \). Now
\[
\left[ (\lambda x. M) N \right]_\rho = (\lambda^G d. \left[ M \right]_{\rho \upharpoonleft x = d}) \cdot F \left[ N \right]_\rho
\]
\[
= F(\lambda d. \left[ M \right]_{\rho \upharpoonleft x = d})(\left[ N \right]_\rho)
\]
\[
= (\lambda d. \left[ M \right]_{\rho \upharpoonleft x = d})(\left[ N \right]_\rho) \quad \text{since } F \cdot G \equiv \text{id},
\]
\[
= \left[ M \right]_{\rho \upharpoonleft x = \left[ N \right]_\rho}.
\]

**Sublemma:** \( \left[ M \left[ x := N \right] \right]_{\rho} = \left[ M \right]_{\rho \upharpoonleft x = \left[ N \right]_\rho} \).

**Subproof:** Induction on the structure of \( M \). Write \( P^* \equiv P \left[ x := N \right], \rho^* = \rho \left[ x := \left[ N \right]_\rho \right] \) and check Table 2.

\[
\begin{array}{|c|c|c|c|}
\hline
M & \left[ M^* \right]_{\rho} & \left[ M \right]_{\rho^*} & \text{Comment} \\
\hline
x & \left[ N \right]_{\rho} & \left[ N \right]_{\rho} & \text{OK} \\
y & \rho(y) & \rho(y) & \text{OK} \\
PQ & \left[ P \right]_{\rho} \cdot \left[ Q \right]_{\rho} & \left[ P \right]_{\rho^*} \cdot \left[ Q \right]_{\rho^*} & \text{IH} \\
\lambda y. P & \lambda^G d. \left[ P \right]_{\rho \upharpoonleft y = d} & \lambda^G d. \left[ P \right]_{\rho^* \upharpoonleft y = d} & (\rho(y = d))^* = \rho^*(y = d) \\
\hline
\end{array}
\]
By the Sublemma, the proof of soundness of (β) is complete. As to (ξ), this was $M = N \Rightarrow \lambda x. M = \lambda x. M$. We have to show

$$D \models M = N \Rightarrow D \models \lambda x. M = \lambda x. M.$$ 

Now

$$D \models M = N \Rightarrow [M]_\rho = [N]_\rho \quad \text{for all } \rho$$

$$\Rightarrow [M]_{\rho(x := d)} = [N]_{\rho(x := d)} \quad \text{for all } \rho, d$$

$$\Rightarrow \lambda d.[M]_{\rho(x := d)} = \lambda d.[N]_{\rho(x := d)} \quad \text{for all } \rho$$

$$\Rightarrow \lambda^G d.[M]_{\rho(x := d)} = \lambda^G d.[N]_{\rho(x := d)} \quad \text{for all } \rho$$

$$\Rightarrow [\lambda x . M]_\rho = [\lambda x . N]_\rho \quad \text{for all } \rho$$

$$\Rightarrow D \models \lambda x . M = \lambda x . N. \quad \square$$

Now we will give an example of a reflexive complete lattice called $D_A$. The method is due to Engeler [32] and is a code-free variant of the graph model $P\omega$ due to Plotkin [75] and Scott [80].

3.2.13. Definition. (i) Let $A$ be a set. Define

$$B_0 = A, \quad B_{n+1} = B_n \cup \{ (\beta, b) \mid b \in B_n \text{ and } \beta \subseteq B_n, \beta \text{ finite} \},$$

$$B = \bigcup_{n} B_n.$$ 

$$D_A = P(B) = \{ x \mid x \subseteq B \},$$

considered as complete lattice under inclusion ($\subseteq$). The set $B$ is just the closure of $A$ under the operation of forming ordered pairs $(\beta, b)$. It is assumed that $A$ consists of urelements, that is, $A$ does not contain pairs $(\beta, b) \in B$.

(ii) Define $F : D_A \rightarrow [D_A \rightarrow D_A], G : [D_A \rightarrow D_A] \rightarrow D_A$ by

$$F(x)(y) = \{ b \mid \exists \beta \subseteq y (\beta, b) \in x \},$$
$$G(f) = \{ (\beta, b) \mid b \in f(\beta) \}.$$ 

3.2.14. Theorem. $D_A$ is reflexive via the maps $F, G$.

Proof. $F$ and $G$ are clearly continuous (use that the $\beta$’s are finite). Moreover, for continuous $f$,

$$F \circ G(f)(y) = F(\{ (\beta, b) \mid b \in f(\beta) \})(y)$$

$$= \{ b \mid \exists \beta \subseteq y b \in f(\beta) \}$$

$$= \bigcup_{\beta \subseteq y} f(\beta)$$

$$= f(y).$$ 

since $\sup = \bigcup$ in $D_A$ and $y = \bigcup_{\beta \subseteq y} \beta$ is a directed supremum. Therefore, $F \circ G(f) = f$ and hence, $F \circ G = \text{id}_{[D_A \rightarrow D_A]}$. \hspace{1cm} \square

Now a semantic proof of the consistency of the lambda calculus can be given.
3.2.15. Corollary. The lambda-calculus is consistent, i.e. $\lambda \vdash \text{true} = \text{false}$.

Proof. Otherwise, $\lambda \vdash x = y$; but then $D_A \vdash x = y$. This is not so; take $\rho(x) \neq \rho(y)$ in a $D_A$ with $A \neq \emptyset$. □

4. Extending the language

In Section 2 we have seen that all computable functions can be expressed in the lambda calculus. For reasons of efficiency, reliability and convenience this language will be extended.

In Section 4.1 some of the constants are selected to represent primitive data (such as numbers) and operations on these (such as addition). Some new reduction rules (the so-called $\delta$-rules) are introduced to express the operational semantics of these operations. Even if these constants and operations can be implemented in the lambda calculus, it is worthwhile to have primitive symbols for them. The reason is that in an implementation of the lambda calculus addition of the Church numerals runs less efficiently than the usual implementation in hardware of addition of binary represented numbers. Having numerals and addition as primitives therefore creates the possibility to interpret these efficiently.

In Section 4.2 types are introduced as a tool towards writing correct software. Types are assigned to terms in various ways using several formal systems. For some of these systems it is decidable whether a term can be typed correctly. This ensures partial correctness of a program.

Before we start Section 4.1, we introduce the following language constructs that are quite useful and are often used in functional programming languages. It allows the programmer to introduce abbreviations for (recursively) defined functions.

4.0. Definition. (i) The expression

"Let $x = M$ in $E"$

stands for "$(\lambda x. E)M$" or "$E[x := M]$". (The latter has some advantages if we want to type expressions: the various occurrences of $M$ may need to be typed differently.)

(ii) The expression

"Letrec $fx = C[f, x] \text{ in } E"$

stands for "Let $f = \Theta(\lambda f\bar{x}. C[f, x]) \text{ in } E"$, where $\Theta$ is Turings fixed point operator (Theorem 3.1.9).

(iii) Similarly, one can have a letrec depending on the double fixed point theorem:

"Letrec $f\bar{x} = C_1[f, g, x]$\n
$g\bar{x} = C_2[f, g, x]$."

These language constructs occur in languages like ML [66], Miranda [92] and TALE [11].
4.1. Delta rules

Let \( X \subseteq A^0 \) be a set of closed normal forms. Usually we take \( X \subseteq C \). Let \( f : X^k \rightarrow A \) be an "externally defined" function. In order to represent \( f \), a so-called \( \delta \)-rule may be added to the lambda calculus. This is done as follows:

1. A special constant in \( C \) is selected and is given some name, say \( \delta (= \delta_f) \).
2. The following contraction rules are added to those of the lambda calculus:

\[
\delta M_1 \ldots M_k \rightarrow f(M_1, \ldots, M_k), \quad M_1, \ldots, M_k \in X.
\]

Note that, for a given function \( f \), this is not one contraction rule but in fact a rule scheme. The resulting extension is called the \( \lambda \delta \)-calculus. The corresponding notion of (one-step) reduction is denoted by \( (\rightarrow_{\beta \delta}) \rightarrow_{\beta \delta} \). So \( \delta \)-reduction is not an absolute notion, but depends on the choice of \( f \).

4.1.1. Theorem (Mitschke). Let \( f \) be a function on closed normal forms. Then the resulting notion of reduction \( \rightarrow_{\beta \delta} \) satisfies the Church–Rosser Theorem.

Proof. This follows from Theorem 15.3.3 in [7]. □

The notion of normal form generalises to \( \beta \delta \)-normal form. So does the concept of leftmost reduction. The \( \beta \delta \)-normal forms can be found by a leftmost reduction.

4.1.2. Theorem. If \( M \rightarrow_{\beta \delta} N \) and \( N \) is in \( \beta \delta \)-nf, then \( M \rightarrow_{\beta \delta} N \).

Proof. Analogous to the proof of Theorem 3.1.10 for \( \beta \)-normal forms. □

4.1.3. Example. One of the first versions of a \( \delta \)-rule can be found in [23]. Here \( X \) is the set of all closed normal forms and, for \( M, N \in X \), we have

\[
\delta_c MN \rightarrow \lambda xy. x \quad \text{if} \ M \equiv N,
\]
\[
\delta_c MN \rightarrow \lambda xy. y \quad \text{if} \ M \not\equiv N.
\]

4.1.4. Exercise. Let \( k_n \) be defined by \( k_0 = k \) and \( k_{n+1} = K(k_n) \). Show that on the \( k_n \) the recursive functions can be represented by terms in \( \lambda \delta_c \).

Another possible set of \( \delta \)-rules is for the Booleans.

4.1.5. Example. The following constants are selected in \( C \).

true, false, not, and, ite (for if then else).

The following \( \delta \)-rules are introduced:

\[
\begin{align*}
\text{not} \ True & \rightarrow \text{false}, \\
\text{and} \ True \ True & \rightarrow \text{true}, \\
\text{and} \ False \ True & \rightarrow \text{false}, \\
\text{ite} \ True & \rightarrow \text{true} \equiv \lambda xy. x, \\
\text{ite} \ False & \rightarrow \text{false} \equiv \lambda xy. y.
\end{align*}
\]
It follows that
\[ \text{ite true } x \ y \rightarrow x, \quad \text{ite false } x \ y \rightarrow y. \]

Now we introduce as \( \delta \)-rules some operations on the set of integers \( \mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \).

4.1.6. Example. For each \( n \in \mathbb{Z} \) a constant in \( C \) is selected and given the name \( n \). (We will express this as follows: for each \( n \in \mathbb{Z} \) a constant \( n \in C \) is chosen.) Moreover, the following constants in \( C \) are selected:

\[ \text{plus, minus, times, divide, error, equal.} \]

Then we introduce the following \( \delta \)-rules (schemes). For \( m, n \in \mathbb{Z} \),

\[ \begin{align*}
\text{plus } n \ m & \rightarrow n + m, \\
\text{minus } n \ m & \rightarrow n - m, \\
\text{times } n \ m & \rightarrow n \times m, \\
\text{divide } n \ m & \rightarrow n \div m \quad \text{if } m \neq 0; \\
\text{divide } n \ 0 & \rightarrow \text{error}, \\
\text{equal } n \ n & \rightarrow \text{true}, \\
\text{equal } n \ m & \rightarrow \text{false} \quad \text{if } n \neq m.
\end{align*} \]

We may add rules like \( \text{plus } n \ \text{error} \rightarrow \text{error} \).

4.1.7. Exercise. Write down a \( \lambda \delta \)-term \( F \) such that \( F \ n \rightarrow n! + n \).

Similar \( \delta \)-rules can be introduced for the set of reals.

Again, another set of \( \delta \)-rules is concerned with characters.

4.1.8. Example. Let \( \Sigma \) be some linearly ordered alphabet. For each symbol \( s \in \Sigma \) we choose a constant \( 's' \in C \). Moreover, we choose two constants \( \delta_\leq \) and \( \delta_\geq \) in \( C \) and formulate the following \( \delta \)-rules:

\[ \begin{align*}
\delta_\leq 's_1' 's_2' & \rightarrow \text{true} \quad \text{if } s_1 \text{ precedes } s_2 \text{ in the ordering of } \Sigma, \\
\delta_\leq 's_1' 's_2' & \rightarrow \text{false} \quad \text{otherwise,} \\
\delta_\geq 's_1' 's_2' & \rightarrow \text{true} \quad \text{if } s_1 = s_2, \\
\delta_\geq 's_1' 's_2' & \rightarrow \text{false} \quad \text{otherwise.}
\end{align*} \]

4.1.9. Exercise. Write down a \( \lambda \delta \)-term \( F \) such that for \( s_1, s_2, t_1, t_2 \in \Sigma \) we have

\[ \begin{align*}
\text{F['s_1', 't_1'][s_2, 't_2'] } & \rightarrow \text{true} \quad \text{if } [s_1, t_1] \text{ precedes } [s_2, t_2] \text{ in the} \\
\text{lexicographical ordering of } \Sigma \times \Sigma, \\
\text{F['s_1', 't_1'][s_2, 't_2'] } & \rightarrow \text{false} \quad \text{otherwise.}
\end{align*} \]

The following is a less orthodox \( \delta \)-rule.

4.1.10. Example. For each integer function \( P \) in Pascal (say with two integer arguments) choose a constant \( \# P \) in \( C \). Add as \( \delta \)-rule

\[ \delta_{\text{Pascal}} \# P \ n \ m \rightarrow k, \]

provided that \( k \) is the value of \( P \) with input \( n \) and \( m \).
It is also possible to represent "multiple value" functions $F$ by putting as $\delta$-rule
$$\delta n \rightarrow m,$$
provided that $F(n) = m.$

Of course, the resulting $\lambda\delta$-calculus does not satisfy the Church–Rosser Theorem and can be used to deal with nondeterministic computations. We will not consider this.

4.2. Types

Types are certain objects, usually syntactic expressions, that may be assigned to terms denoting programs. Types serve as a classification of the (objects denoted by the) terms. Each type $\sigma$ has as semantics a set $D_{\sigma}$ of "objects of type $\sigma$". There are several systems of type assignment with different collections of types. We will consider a few of these. (For the more complicated type systems the semantics, $D_{\sigma}$ will in general be not a set, but an object in some category.)

Type assignment is done for the following reasons. Firstly, the type of a term $F$ gives a partial specification of what the function denoted by $F$ is supposed to do. Usually, this type a specification is given before the term as program is constructed. The verification whether this term, once it has been constructed, is indeed of the required type provides a partial correctness proof for the program. Secondly, types play a role in efficiency. If it is known that a subterm $S$ of a program has a certain type, then $S$ may be executed more efficiently by making use of the type information.

To explain the idea of type assignment, we present type systems of various strengths. We start with a simple set of types $\text{Type}_1$, inductively defined as follows.

4.2.1. Definition

$$\text{B} \in \text{Type}_1, \quad \text{Z} \in \text{Type}_1, \quad \text{Char} \in \text{Type}_1 \quad \text{(type constants),}$$

$$\sigma \in \text{Type}_1, \tau \in \text{Type}_1 \quad \Rightarrow (\sigma \rightarrow \tau) \in \text{Type}_1.$$

Convention. \(\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n\) stands for \((\sigma_1 \rightarrow (\sigma_2 \rightarrow \cdots (\sigma_{n-1} \rightarrow \sigma_n)))\).

The intuitive semantics for these types is as follows. The types $\text{Z}$, \text{Char} and \text{B} are used to denote respectively the sets of integers, characters and Booleans. That is, $D_{\text{Z}} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ etcetera. $D_{\sigma \rightarrow \tau} = D_{\tau}^{D_{\sigma}}$, i.e. the set of functions from $D_{\sigma}$ to $D_{\tau}$. (For more complicated systems $D_{\sigma \rightarrow \tau}$ is only a subset or subobject of $D_{\tau}^{D_{\sigma}}$.)

Now we will see how types are used in the form of so called type assignment.

4.2.2. Definition. Let $\text{Type}$ be a set of types.

(i) A statement is of the form $M: \sigma$, where $\sigma \in \text{Type}$ and $M$ is a $\lambda\delta$-term. $M$ is called the subject and $\sigma$ the predicate of the statement.

(ii) A theory of type assignment consist of a set of statements axiomatised in some way.

4.2.3. Definition. Consider the $\lambda\delta$-terms for Booleans, integers and characters introduced in Examples 4.1.3–5. Consider the set $\text{Type}_1$ introduced in Definition 4.2.1. The theory of type assignment $T_1$ is defined by the following set of axioms and rules.
Axioms:

1. true: B, false: B, not: B → B, and: B → B → B,
2. n: Z; error: Z,
   plus, minus, times, divide: Z → Z → Z,
   equal: Z → Z → B,
3. δ_<, δ_>: Char → Char → B.

Rule:
4. If M: σ → τ and N: σ, then MN: τ.

The system $T_1$ is very weak. We can only derive statements like equal (plus 3 4) 12: B. No types more complex than the ones given in the axioms are possible in type assignment. In the following system $T_2$, essentially due to Curry, more statements can be derived.

4.2.4. Definition. The set of types Type$ _2$ is inductively defined as follows:

\[
\begin{align*}
\text{BeType}_2, \ &Z \in \text{Type}_2, \ &\text{Char} \in \text{Type}_2, \\
\alpha_0, \alpha_1, \ldots \in \text{Type}_2 \ &\text{(type variables),} \\
\sigma, \tau \in \text{Type}_2 \ &\Rightarrow (\sigma \rightarrow \tau) \in \text{Type}_2.
\end{align*}
\]

The $\alpha_0, \alpha_1, \ldots$ are called type variables and are often denoted by $\alpha, \beta, \gamma, \ldots$.

The interpretation of types now becomes dependent on a valuation $\xi$ that assigns to type variables some set (or object in some category) $\xi(\alpha)$:

\[
\begin{align*}
D^\xi_2 &= \xi(\alpha), \\
D^\xi_Z &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \ etcetera, \\
D^\xi_{\alpha \rightarrow \tau} &= D^\xi(\alpha) \rightarrow D^\xi(\tau).
\end{align*}
\]

In order to give the system for type assignment we need some more concepts.

4.2.5. Definition. A basis is a set $B$ of statements in which the subjects are distinct term variables. The notion that a statement $M: \sigma$ is derivable from a basis $B$, notation $B \vdash M: \sigma$, is defined inductively as follows:

Axioms:

0. $x: \sigma \in B \Rightarrow B \vdash x: \sigma,$
1. $B \vdash \text{true}: B, \quad B \vdash \text{false}: B,$
   $B \vdash \text{not}: B \rightarrow B, \quad B \vdash \text{and}: B \rightarrow B \rightarrow B,$
   $B \vdash n: Z; \quad B \vdash \text{error}: Z,$
   $B \vdash \text{plus, minus, times, divide}: Z \rightarrow Z \rightarrow Z,$
   $B \vdash \text{equal}: Z \rightarrow Z \rightarrow B$
   $B \vdash \delta_<, \delta_> \colon \text{Char} \rightarrow \text{Char} \rightarrow B.$

Rules:

2. $B \vdash M: \sigma \rightarrow \tau, \ B \vdash N: \sigma \Rightarrow B \vdash MN: \tau,$
The axioms and rules can be summarised in a convenient natural deduction notation as follows:

**Axioms:**

1. true: B  
   false: B  etcetera.

(The axiom (0) is not written down in this notation, but implicitly understood.)

**Rules:**

2. \[ M : \sigma \rightarrow \tau \quad N : \sigma \]  
   \[ \frac{}{MN : \tau} \]

3. \[ [x : \sigma] \]
   \[ \frac{}{M : \tau} \]
   \[ \lambda x. M : \sigma \rightarrow \tau \]

**Example.** In \( T_2 \) we can derive the following statements. We write \( \vdash M : \sigma \) for \( \emptyset \vdash M : \sigma \).

\[ \vdash \lambda xy.x : \sigma \rightarrow (\tau \rightarrow \sigma) \quad y : \tau \vdash \lambda x. y : \sigma \rightarrow \tau, \]

\[ \vdash (\lambda xyz. \times x (\text{plus } y z)) : Z \rightarrow Z \rightarrow Z \rightarrow Z. \]

A derivation for the first statement looks like

\[ \begin{array}{c}
   [x : \sigma] \\
   [y : \tau] \\
   x : \sigma \\
   \frac{}{\lambda y. x : \tau \rightarrow \sigma} \\
   \lambda x y. x : \sigma \rightarrow (\tau \rightarrow \sigma)
\end{array} \]

Note that in this derived statement \( \sigma \) and \( \tau \) are arbitrary types.

If \( T \) is a system of type assignment, then a term \( M \) is called typable if for some \( B \) and \( \sigma \) one has \( B \vdash M : \sigma \). The following result is independently due to Curry [28] and Hindley [41] and was rediscovered by Milner [65].

**4.2.6. Theorem.** (i) It is decidable whether a term is typable in \( T_2 \).

(ii) If a term \( M \) has a type in \( T_2 \), then \( M \) has a unique principal type scheme, i.e. a type \( \sigma \) such that every possible type for \( M \) is a substitution instance of \( \sigma \). Moreover, \( \sigma \) is computable from \( M \).

For example the principal type scheme of \( \lambda xy.x \) is \( \alpha \rightarrow (\beta \rightarrow \alpha) \).

The following result states that terms typable in \( T_2 \) all have a normal form.

**4.2.7. Theorem.** In \( T_2 \) we have \( B \vdash M : \sigma \Rightarrow M \) has a normal form.

**Proof.** This follows from the same property for \( T_3 \), see Theorem 4.2.13. □

The following result is called the "subject reduction theorem" by Curry and follows inductively from the rules of type assignment.
4.2.8 Theorem. The following holds for $T_2$

$$B \vdash M : \sigma \quad \text{and} \quad M \rightarrow M' \Rightarrow B \vdash M' : \sigma.$$  

The type assignment system $T_2$ is still rather poor. For example, the term \texttt{ite} representing the conditional has no natural type. For this reason we will extend the set of types and the rules for type assignment. The resulting system is a variant (the "Curry version" in the terminology of [8]) of the polymorphic second-order lambda calculus independently due to [37] and [78].

4.2.9. Definition. Type $3$ is the set of types inductively defined by

$$B \in \text{Type}_3, \quad Z \in \text{Type}_3, \quad \text{Char} \in \text{Type}_3, \quad \alpha_0, \alpha_1, \ldots \in \text{Type}_3,$$

$$\sigma \in \text{Type}_3, \tau \in \text{Type}_3 \Rightarrow (\sigma \rightarrow \tau) \in \text{Type}_3,$$

$$\sigma \in \text{Type}_3 \Rightarrow \forall \alpha, \sigma \in \text{Type}_3.$$

4.2.10. Definition. $T_3$ is the system of type assignment with types in $\text{Type}_3$ defined by the following axioms and rules (in a natural deduction formulation).

(1) true: $B$, false: $B$, not: $B \rightarrow B$, and: $B \rightarrow B \rightarrow B$, \n
\[ n: Z \quad \text{error: } \forall \alpha, \alpha, \] \n
plus, minus, times, divide: $Z \rightarrow Z \rightarrow Z$, \n
equal: $Z \rightarrow Z \rightarrow B$, \n
$\delta_>, \delta_-=: \text{Char} \rightarrow \text{Char} \rightarrow B$, \n
\text{ite: } \forall \alpha, B \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha.$

(2) $M: \sigma \rightarrow \tau$ \hspace{1cm} $N: \sigma$ \hspace{1cm} $\vdash M \cdot N : \tau$

(3) \hspace{1cm} $\vdash x: \sigma$

(4) $\vdash M: \forall \alpha, \sigma$ \hspace{1cm} \hspace{1cm} $\vdash M: \sigma \rightarrow \tau$

(5) $\vdash M: \sigma$ \hspace{1cm} $\vdash M: \forall \alpha, \sigma$

In rule (5) the type variable may not occur in the assumptions on which $M: \sigma$ depends. That is, rule (5) should be read as follows:

$$B \vdash M: \sigma, \alpha \text{ not free in } B \Rightarrow B \vdash M: \forall \alpha, \sigma.$$  

4.2.11. Examples. In $T_3$ the following statements can be derived:

(i) $\vdash \text{-1: } \forall \alpha, \alpha \rightarrow \alpha$;

(ii) Define $\text{Nat} = \forall \alpha, \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. Then for the Church numerals $c_n \equiv \lambda f. f^n x$ we have $\vdash c_n: \text{Nat}$.

This example shows the intended meaning of the types $\forall \alpha, \sigma$. For example in $T_2$ we had $\vdash \text{-1: } \sigma \rightarrow \sigma$ for all $\sigma$. Now in $T_3$ we have $\vdash \text{-1: } \forall \alpha, \alpha \rightarrow \alpha$. A formal semantics for $T_3$ gives rise to interesting (categorical) notions and will not be discussed here (see [37, 82, 48].
On the Church numerals, many computable functions can be represented by terms of type \( \text{Nat} \rightarrow \text{Nat} \). The next result is due to [36]; see also [34].

**4.2.12. Theorem.** A numeric function \( f \) is representable by a term of type \( \text{Nat} \rightarrow \text{Nat} \) iff \( f \) is an in analysis provable total recursive function.

The following normalisation result is due to [36]. For a proof see also [85]. The subject reduction theorem is also valid.

**4.2.13. Theorem.** In \( T_3 \) we have

(i) \( B \vdash M \sigma \Rightarrow M \) has a normal form;
(ii) \( B \vdash M \sigma \) and \( M \rightarrow M' \Rightarrow B \vdash M' \sigma \).

**4.2.14. Open Problem.** Is it decidable whether a term has a type in \( T_3 \)?

So far all systems of type assignment are such that typable terms have a normal form. It will now be shown that type assignment systems with this property have as limitation that not all computable functions are representable by a typed term.

**4.2.15. Theorem.** Let \( \lambda \delta \) be an extension of the lambda calculus by means of effective \( \delta \)-rules \( (M \rightarrow_{\text{pd}} N \text{ should be an r.e. relation in } M, N) \). Let \( T \) be an effective system of type assignment \( (\leftarrow M \sigma \text{ should be an r.e. relation in } M) \) such that every typable term has a normal form. Then not all computable functions are representable in \( \lambda \delta \) by a term typable in \( T \).

**Proof.** Recall that a numeric function \( F(n, m) \) is called universal for a class of unary numeric functions \( \mathcal{A} \) if \( \mathcal{A} = \{ f_0, f_1, \ldots \} \), where \( f_n \) is defined by \( f_n(m) = F(n, m) \).

Let \( \mathcal{A}_k \) be the class of total numeric computable functions with \( k \) arguments. It is well-known that there is no \( F \in \mathcal{A}_2 \) that is universal for \( \mathcal{A}_1 \). (Otherwise the function \( g(n) = F(n, n) + 1 \) will yield a contradiction.) It follows that not all computable functions are representable by a typable term. (Otherwise one could obtain a computable function universal for \( \mathcal{A}_1 \) by enumerating terms: define

\[
F(n, m) = k \iff \text{the } n\text{th typed term (say with type } \text{Z} \rightarrow \text{Z}) \text{ applied to the argument } m \text{ has as normal form } k.
\]

In order to construct this \( F \), we need that reduction and typing are effective.)

The following system of type assignment is such that all computable functions are representable by a typed term. Indeed, the system also assigns types to nonnormalising terms by introducing a primitive fixed point combinator \( Y \) having type \( \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \).

**4.2.16. Definition.** (i) The \( \lambda Y \delta \)-calculus is an extension of the \( \lambda \delta \)-calculus in which there is a constant \( Y \) with reduction rule \( Yf \rightarrow f(Yf) \).

(ii) \( T_4 \) is the system that assigns types in \( \text{Type}_3 \) to \( \lambda Y \delta \)-terms. \( T_4 \) is defined by adding to the system \( T_3 \) the axiom

\[
Y : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha.
\]
Because of the presence of $Y$, not all terms have a normal form. Without proof we state the following theorem.

**4.2.17. Theorem.**

(i) The $\lambda Y\delta$-calculus satisfies the Church–Rosser property.

(ii) If a term in the $\lambda Y\delta$-calculus has a normal form, then it can be found using leftmost reduction.

(iii) The subject reduction theorem holds for $T_4$.

**4.2.18. Theorem.** All computable functions can be represented in $\lambda Y\delta$ by a term typable in $T_4$.

**Proof.** The construction uses the primitive numerals $n$. If we take $\text{zero} \equiv \lambda x.\text{equal} 0 x$ and $\text{P}^- \equiv \lambda x.\text{minus} x 1$, then the proof of Theorem 2.2.13 can be imitated using $Y$ instead of the fixed point combinator. The types for the functions defined using $Y$ are natural. $\square$

Some relevant articles are: [6, 12, 14, 16, 20, 21, 24, 25, 35, 43, 48, 61, 62, 64, 70, 81, 82].

The following volumes contain many articles on typed lambda calculus: [51] and the Proceedings of the Conferences on Logic in Computer Science (IEEE, 1986, 1987 and 1988). See also the chapter on type systems for programming languages by Mitchell in this Handbook, [8] and [38].

Many aspects of types have been omitted in this section, notably

1. types for (Cartesian) products, (disjoint) unions and intersections,
2. recursive types,
3. dependent types,
4. type inclusion,
5. abstract data types,
6. formulae as types,
7. semantics: for simple, polymorphic and dependent types, fully abstract models.

Some relevant articles are: [6, 12, 14, 16, 20, 21, 24, 25, 35, 43, 48, 61, 62, 64, 70, 81, 82]. The following volumes contain many articles on typed lambda calculus: [51] and the Proceedings of the Conferences on Logic in Computer Science (IEEE, 1986, 1987 and 1988). See also the chapter on type systems for programming languages by Mitchell in this Handbook, [8] and [38].

5. The theory of combinators and implementation issues

5.1. The theory of combinators

In the lambda calculus the notation of bound variable is used. This does not only cause some complications in the theory (avoiding clashes of bound and free variables) but also in implementations of the lambda calculus. On way to avoid these problems is to translate the lambda calculus into systems without free variables, the so-called combinatory systems.
5.1.1. Definition. The set of combinatory terms, notation \( \mathcal{C} \), is inductively defined as follows.

\[
\begin{align*}
    c \in C & \Rightarrow c \in \mathcal{C}, \\
    x \in V & \Rightarrow x \in \mathcal{C}, \\
    P, Q \in \mathcal{C} & \Rightarrow (PQ) \in \mathcal{C}.
\end{align*}
\]

where \( C \) and \( V \) are the sets of constants and variables respectively, as in Definition 2.1.1.

5.1.2. Definition. (i) The theory \( \text{CL}(I, K, S) \) has as terms the set \( \mathcal{C} \). Among the constant of \( \mathcal{C} \) three are selected and given the names \( I, K \) and \( S \) respectively. The theory is axiomatised by the following set of axioms:

\[ IP = P, \quad KPQ = P, \quad SPQR = PR(QR). \]

The rules are just the rules of equality and are the same as in Definition 2.1.5(ii), except that the rule (\( \xi \)) is not present. We still usually write \( \text{CL} \) for \( \text{CL}(I, K, S) \).

(ii) The axioms in (i) correspond to a relation of weak reduction by setting

\[ IP \rightarrow_w P, \quad KPQ \rightarrow_w P, \quad SPQR \rightarrow_w PR(QR). \]

The relation \( \rightarrow_w \) which is compatible with application generates just as in Definition 3.1.2 the reduction relation \( \rightarrow^w \) and conversion relation \( =_w \). Similarly to Proposition 3.1.3 we have

\[ \text{CL} \vdash P = Q \iff P =_w Q. \]

(iii) The \( \delta \)-rules discussed in Section 4.1 can be added to the theory \( \text{CL}(I, K, S) \), obtaining \( \text{CL}(I, K, S, \delta) \) or simply \( \text{CL}\delta \).

5.1.3. Definition. (i) For we define \( \lambda^* x . P \in \mathcal{C} \) to simulate abstraction in \( \text{CL} \).

\[ \lambda^* x . x = I, \quad \lambda^* x . P = KP \text{ if } x \text{ does not occur in } P, \quad \lambda^* x . PQ = S(\lambda^* x . P)(\lambda^* x . Q). \]

(ii) Lambda terms can be translated into combinatory terms: for \( M \in A \) we define \( M_{\text{CL}} \in \mathcal{C} \) as follows:

\[
\begin{align*}
    c_{\text{CL}} & \equiv c; \\
    x_{\text{CL}} & \equiv x; \\
    (MN)_{\text{CL}} & \equiv M_{\text{CL}}N_{\text{CL}}; \\
    (\lambda x . M)_{\text{CL}} & \equiv \lambda^* x . M_{\text{CL}}.
\end{align*}
\]

5.1.4. Lemma. For \( P, Q, \in \mathcal{C} \) and \( M, N \in A \) we have

(i) \( (\lambda^* x . P)Q \rightarrow_w P[x := Q] \),

(ii) \( (\lambda^* x . P)[y := Q] \equiv \lambda^* x . (P[y := Q]) \),

(iii) \( M[x := N]_{\text{CL}} \equiv M_{\text{CL}}[x := N_{\text{CL}}] \)

Proof. (i), (ii) By induction on the structure of \( P \).

(iii) By induction on the structure of \( M \), using (ii). \( \square \)
There is a difference between $\beta$-reduction and $w$-reduction. $M_{CL}$ may be a nf for $w$-reduction but $M$ itself not for $\beta$-reduction. Take, e.g., $M \equiv \lambda x.Ix$ with $M_{CL} \equiv S(KI)I$. Therefore $CL$ does not seem to be a good intermediate language for evaluating $\lambda$-terms. We will show now that for terms having a ground type the normal form can be found via translation and reduction in $CL$.

5.1.5. Lemma. (i) Let $M \rightarrow_{\beta} N$ with $M$ closed and not of the form $\lambda x.M$; then $M_{CL} \rightarrow_{w} N_{CL}$.

(ii) Let $\rightarrow_{\beta} N$ with $M$ closed and typable in $T_4$ with a ground type ($Z$, $B$ or $Char$); then $M_{CL} \rightarrow_{w} N_{CL}$.

Proof. (i) Induction on the shape of $M$. If $M \equiv (\lambda x.M_0)M_1L$, then the result follows from Lemma 5.1.4. If $M = cL_1, \ldots, L_k$ and $N = cL'_1, \ldots, L'_k$, then the induction hypothesis applies to $L_i$. If $M = cL$ is a $\delta$-redex, then its $CL$-translation is a $\delta$-redex too.

(ii) If $M$ has a ground type, then so does each reduct $M'$ of $M$ and hence $M' \not\equiv \lambda x.M'$. The result follows by repeated application of (i). □

5.1.6. Corollary. Let $\lambda \delta \vdash M = c$ with $c$ a constant and $M$ typable in $T_4$ with a ground type. Then $CL\delta \vdash M_{CL} = c$.

Proof. $\lambda \delta \vdash M = c \Rightarrow M = \beta \delta c$

$\Rightarrow M \rightarrow_{\beta} c$

$\Rightarrow M_{CL} \rightarrow_{w} c_{CL} \equiv c$ by Lemma 5.1.5(ii)

$\Rightarrow CL\delta \vdash M_{CL} = c$. □

An important consequence is that in order to compute the $\lambda$-terms it is sufficient to build an implementation for combinatory reduction as long as the output is of a basic type. Indeed, many reduction machines are based on a combinator implementation (see, e.g., [90]).

5.2. Implementation issues

Implementing functional languages has become an art in itself. In this subsection we briefly discuss some issues and refer the reader to the literature. In particular to the comprehensive [74] and also to the conference proceedings [26, 50, 5, 52].

LISP

LISP is a language that is not exactly functional but has several features of it. Because implementations of several variants of LISP are quite successfully used, we want to indicate why we prefer functional languages. This preference explains why research is being done to make also fast implementations for these languages.

The following features of LISP 1.5 [63] make it different from functional languages:

1. There are assignment and goto statements.
2. There is "dynamic binding", which is a wrong kind of substitution in which a free variable may become bound.
3. There is
a "quote" operator that "freezes" the evaluation of subterms. This quote is not referentially transparent, since quote \((Ic)\neq quote c\).

In modern versions of LISP, e.g. SCHEME [84], there are no more goto statements and dynamic binding has been replaced by "static binding", i.e. the proper way to do substitution. However in SCHEME one still uses assignments and the quote operator.

Since there is no assignment in functional languages, there is a more clear semantics and therefore we have easier correctness proofs. For the same reason parallel computing is more natural using functional languages. Also functional languages have strong typing, something that is missing in LISP. The quote is sometimes thought to be essential to construct the "metacircular" interpreter for LISP written in LISP. However, self-interpretation is also possible in functional programming, see Theorem 2.2.20.

Another aspect of LISP is that the quote may be used in order to write "self-modifying" programs. This is also possible in the lambda calculus, even if the quote is not \(\lambda\)-definable. Suppose that we want a typically self-modifying function \(F\), e.g.,

\[
Fx = \text{if } Px \text{ then } G_1x \text{ else } G_2x F^\gamma x,
\]

where \(P\), \(G_1\) and \(G_2\) are given and \(F^\gamma\) is the code of \(F\) represented as numeral in the lambda calculus. Then such an \(F\) can be found as a \(\lambda\)-term using the second Fixed Point Theorem 2.2.24:

\[
F = (\lambda fx. \text{if } Px \text{ then } G_1x \text{ else } G_2fx) F^\gamma.
\]

So with lambda calculus we can do in a more hygienic way most things that can be done with LISP. Nevertheless, LISP is an important language. Experience gained by using and implementing LISP has been useful in functional languages.

The SECD-machine

The first proposal to implement a functional language was done by Landin [57]. He introduced the so called SECD-machine. It uses the idea of an environment, i.e. a list of pairs of variable an associated value that is dynamically updated. The SECD-machine can be used to support both eager and lazy evaluation [17]. A modern version of the SECD-machine is the CAM (Categorical Abstract Machine) [27].

Other combinators

The \(\lambda^*\)-algorithm to imitate abstraction in the combinatory system is rather inefficient. If \(M\) if of length \(n\), then the length of \(M_{\text{CL}}\) is \(O(n^2)\). Turner [90] gives an improved algorithm in which \(M_{\text{CL}}\) is \(O(n^2)\). Statman [83] gives an algorithm with \(M_{\text{CL}}\) of length \(O(n \log n)\) and shows that this is indeed best possible. However, his constant factor is such that the algorithm of Turner seems to be better in practical cases. In [53, 69] the complexity of various translations into fixed sets or families of combinators are compared.

Another possibility is to translate \(\lambda\)-terms to "user-defined combinators". For example a \(\lambda\)-term like

\[
\lambda xyz. x(\lambda ab. ay)z w
\]
may be translated into $A w$ with the ad hoc rules

$$A w x y z \rightarrow x (B y z) w, \quad B y z a b \rightarrow a y z.$$  

This approach is taken in [47] (supercombinators), [44] (serial combinators) and [49] (lambda lifting).

**Term rewrite systems**

In the lambda calculus we have ordered pairs and projections. Also other constructors and selectors are definable. Introducing primitive constants for these allows a more efficient implementation. For example we may select some constants $p, p_1, p_2$ and add the rules

$$p_1(p x y) \rightarrow x, \quad p_2(p x y) \rightarrow y.$$  

These kind of rules are typical members of so called term rewrite systems (TRS). These TRS’s are quite flexible to describe what a functional program should do. For example, a (double) fixed point combinator does not require a theorem, but can be written down immediately. See [46, 55, 56] for an exposition of the subject and [72] for the use of TRS’s in programming.

**Graph reduction**

If a variable occurs more than once in a term $M (≡ M[x, x]$, say), then the reduction

$$(\lambda x . M[x, x]) N \rightarrow M[N, N]$$

with the actual substitution of $N$ in $M$ results in an increase of complexity. In [93] it was proposed that instead of performing the actual substitution, the contraction may be represented as a graph (see Fig. 4) with two pointers from within $M$ towards one occurrence of $N$. This saves not only space but also execution time, since contraction within $N$ now needs to be done only once, not twice. In [90] another use of graphs is proposed. For example, the $Y$ combinator with contraction rule $Y f \rightarrow f(Y f)$ can be presented using a cyclic graph (see Fig. 5). Cyclic graphs do speed up performance of implemented functional languages; however, garbage collection becomes more complicated.

![Fig. 4](image4.png)

![Fig. 5](image5.png)
In [9] theorems are proved relating ordinary reduction to graph reduction. In [15] the language Clean based on term rewrite systems and graph reduction is proposed as an intermediate between functional languages and reduction machines.

**Strictness analysis**

A function $F$ is said to be *strict* if for the evaluation of $FA$ the argument $A$ is always needed. For instance $I = \lambda x . x$ is strict, but $K0 = \lambda x . 0$ is not. Knowing which parts of a functional program are strict is useful for a compiler, since then eager evaluation may be used for those parts, hence obtaining a performance improvement. Although the notion of strictness is in general undecidable, there are some interesting decidable approximations of this notion. See [45, 71, 18, 10, 77] for work in this direction.

Information about types and strictness are important for obtaining speed optimisations. This is explained in [74] and exploited in e.g. the implementations described in [3, 49, 15].

**Modifications of the lambda-calculus**

Many implementations of functional languages are such that expressions like $\lambda x . M$ are not reduced any further. For this reason Abramsky introduced the notion of "lazy lambda calculus" in order to make the theory correspond better to this practice, see [73]. Also [76] (call-by-name lambda calculus) and [67] (partial lambda calculus) are relevant in this respect.

**Acknowledgment**

I would like to thank Mariangiola Dezani-Ciancaglini, Roger Hindley, Bart Jacobs and Pieter Koopman for useful suggestions on a draft version of this chapter.

**References**


[75] Plotkin, G., A set-theoretical definition of application, Memo MIP-R-95, School of Artificial Intelligence, Univ. of Edinburgh, 1972.


